Boosting Model Checking to Analyse Large ARBAC Policies

Silvio Ranise[1], Anh Truong[1], and Alessandro Armando[1,2]

[1] Security and Trust Unit, FBK-Irst, Trento, Italia
[2] DIST, Università degli Studi di Genova, Italia

Abstract. The administration of access control policies is a task of paramount importance for distributed systems. A crucial analysis problem is to foresee if a set of administrators can give a user an access permission. We consider this analysis problem in the context of the Administrative Role-Based Access Control (ARBAC), one of the most widespread administrative models. Given the difficulty of taking into account the effect of all possible administrative actions, automated analysis techniques are needed. In this paper, we describe how a model checker can scale up to handle very large ARBAC policies while ensuring completeness. An extensive experimentation shows that an implementation of our techniques performs significantly better than MOHAWK, a recently proposed tool that has become the reference for finding errors in ARBAC policies.

1 Introduction

The administration of access control policies is a task of paramount importance for the flexibility and security of many distributed systems. For flexibility, administrative actions are carried out by several security officers. For security, the capabilities of performing such operations must be limited to selected parts of the access control policies since officers can only be partially trusted. Indeed, flexibility and security are opposing forces and avoiding under- or over-constrained specifications of administrative actions is of paramount importance. In this respect, it is crucial to foresee if a user can get a certain permission by a sequence of administrative actions executed by a set of administrators. Since it is difficult to take into account the effect of all possible administrative actions, push-button analysis techniques are needed.

Role-Based Access Control (RBAC) [12] is one of the most widespread authorization model and Administrative RBAC (ARBAC) [5] is the corresponding widely used administrative model. In RBAC, access control policies are specified by assigning users to roles that in turn are assigned to permissions. ARBAC allows for the specification of rules that permit to modify selected parts of a RBAC policy. The analysis problem consists of establishing if a certain user can be assigned to a certain role (or permission) by a sequence of administrative actions. Several automated analysis techniques (see, e.g., [11, 10, 14]) have been developed for solving this problem in the ARBAC model. Recently, a tool called ASASP [2] has been shown (in [4]) to perform better than the state-of-the-art tool

RBACPAT [8] on the set of benchmark problems in [14] and (in [3]) to be able to tackle more expressive ARBAC policies that RBACPAT cannot handle. More recently, another tool called MOHAWK has been shown (in [9]) to scale much better than RBAC-PAT on the problems in [14] and more complex ones.

In this paper, we investigate how the model checking techniques underlying ASASP can scale up to solve the largest problem instances in [9]. In fact, preliminary experiments showed that ASASP can only handle instances of moderate size in [9]. This has lead us to develop a new version of ASASP, called ASASPXL, with the goal of boosting the model checking techniques underlying ASASP and guaranteeing to find errors in ARBAC policies if they exist. This is in contrast with the approach of MOHAWK, which—as said in [9]— is incomplete, i.e. it may miss errors in a buggy policy. An extensive experimental comparison on the benchmark problems in [9] shows that ASASPXL performs significantly better than MOHAWK.

Plan of the paper. Section 2 introduces the ARBAC model and the related analysis problem. Section 3 briefly reviews the model checking technique underlying ASASP. Section 4 describes the techniques that we have designed for scalability. Section 5 summarizes the findings of our experiments. Section 6 concludes.

2 Administrative Role-Based Access Control

In *Role-Based Access Control (RBAC)* [12], access decisions are based on the roles that individual users have as part of an organization. Permissions are grouped by role name and correspond to various uses of a resource. Roles can have overlapping responsibilities and privileges, i.e. users belonging to different roles may have common permissions. To allow for compact specifications of RBAC policies, role hierarchies are used to reflect the natural structure of an enterprise and make the specification of policies more compact by requiring that one role may implicitly include the permissions that are associated with others.

RBAC policies need to be maintained according to the evolving needs of the organization. For flexibility and scalability, large systems usually require several administrators, and thus there is a need not only to have a consistent RBAC policy but also to ensure that the policy is modified by administrators who are allowed to do so. Several administrative frameworks have been One of the most popular administrative frameworks is Administrative RBAC (ARBAC) [5] that controls how RBAC policies may evolve through administrative actions that assign or revoke user memberships into roles. Since administrators can be only partially trusted, administration privileges must be limited to selected parts of the RBAC policies, called *administrative domains*. The ARBAC model defines administrative domains by using RBAC itself to control how security officers can delegate (part of) their administrative permissions to trusted users. Despite such restrictions, it is very difficult to foresee if a subset of the security officers can maliciously (or inadvertently) assign a role to an untrusted user that enable him/her to get access to security-sensitive resources.

Formalization. Let U be a set of users, R a set of roles, and P a set of permissions. Users are associated to roles by a binary relation $UA \subseteq U \times R$ and roles are associated to permissions by another binary relation $PA \subseteq R \times P$. A role hierarchy is a partial order $\succeq$ on R, where $r_1 \succeq r_2$ means that r_1 is *more senior than* r_2 for $r_1, r_2 \in R$. A user u is a *member* of role r when $(u, r) \in UA$. A user u *has permission* p if there exists a role $r \in R$ such that $(p, r) \in PA$ and u is a member of r. A *RBAC policy* is a tuple $(U, R, P, UA, PA, \succeq)$.

Usually (see, e.g., [14]), administrators may only update the relation UA while PA and $\succeq$ are assumed constant. An administrative domain is specified by a *pre-condition*, i.e. a finite set of expressions of the forms r or $\overline{r}$ (for $r \in R$). A user $u \in U$ *satisfies* a pre-condition C if, for each $\ell \in C$, u is a member of r when ℓ is r or u is not a member of r when ℓ is $\overline{r}$ for $r \in R$. Permission to assign users to roles is specified by a ternary relation *can_assign* containing tuples of the form (C_a, C, r) where C_a and C are pre-conditions, and r a role. Permission to revoke users from roles is specified by a binary relation *can_revoke* containing tuples of the form (C_a, r) where C_a is a pre-condition and r a role. In both cases, we say that C_a is the *administrative pre-condition*, C is a *(simple) pre-condition*, r is the *target role*, and a user u_a satisfying C_a is the *administrator*. When there exist users satisfying the administrative and the simple (if the case) pre-conditions of an administrative action, the action is *enabled*. The relation *can_revoke* is only binary because simple pre-conditions are useless when revoking roles (see, e.g., [14]). The semantics of the administrative actions in $\psi := (\mathit{can\_assign}, \mathit{can\_revoke})$ is given by the binary relation $\rightarrow_\psi$ defined as follows: $UA \rightarrow_\psi UA'$ iff there exist users u_a and u in U such that either (i) there exists $(C_a, C, r) \in \mathit{can\_assign}$, u_a satisfies C_a, u satisfies C (i.e. (C_a, C, r) is enabled), and $UA' = UA \cup \{(u, r)\}$ or (ii) there exists $(C_a, r) \in \mathit{can\_revoke}$, u_a satisfies C_a (i.e. (C_a, r) is enabled), and $UA' = UA \setminus \{(u, r)\}$. A *run* of the administrative actions in $\psi := (\mathit{can\_assign}, \mathit{can\_revoke})$ is a possibly infinite sequence $UA_0, UA_1, ..., UA_n, ...$ such that $UA_i \rightarrow_\psi UA_{i+1}$ for $i \geq 0$.

A pair (u_g, R_g) is called a *(RBAC) goal* for $u_g \in U$ and R_g a finite set of roles. The cardinality $|R_g|$ of R_g is the *size* of the goal. Given an initial RBAC policy UA_0, a goal (u_g, R_g), and administrative actions $\psi = (\mathit{can\_assign}, \mathit{can\_revoke})$; (an instance of) the *user-role reachability problem*, identified by the tuple $\langle UA, \psi, (u_g, R_g)\rangle$, consists of checking if there exists a finite sequence $UA_0, UA_1, ..., UA_n$ (for $n \geq 0$) where (i) $UA_i \rightarrow_\psi UA_{i+1}$ for each $i = 0, ..., n-1$ and (ii) u_g is a member of each role of R_g in UA_n.

The user-role reachability problem defined here is the same of that in [14, 9]. In the rest of the paper, we focus on problem instances where U and R are finite, P plays no role, and $\succeq$ can be ignored (see, e.g., [13]). Thus, a RBAC policy is a tuple (U, R, UA) or simply UA when U and R are clear from the context.

3 Model Checking Modulo Theories and ARBAC Policies

Prologue. Model Checking Modulo Theories (MCMT) [7] is a framework to solve reachability problems for infinite state systems that can be represented by transition systems whose set of states and transitions are encoded as constraints in first-order logic. Such symbolic transition systems have been used as abstractions of parametrised protocols, sequential programs manipulating arrays, timed system, etc (see again [7] for an overview).

The main idea underlying the MCMT framework is to use a backward reachability procedure that repeatedly computes pre-images of the set of *goal* states, that is usually obtained by complementing a certain safety property that the system should satisfy. The set of backward reachable states of the system is obtained by taking the union of the pre-images. At each iteration of the procedure, it is checked whether the intersection with the initial set of states is non-empty (*safety* test) and the *unsafety* of the system (i.e. there exists a (finite) sequence of transitions that leads the system from an initial state to one satisfying the goal) is returned. Otherwise, when the intersection is empty, it is checked if the set of backward reachable states is contained in the set computed at the previous iteration (*fix-point* test) and the *safety* of the system (i.e. no (finite) sequence of transitions leads the system from an initial state to one satisfying the goal) is returned. Since sets of states and transitions are represented by first-order constraints, the computation of pre-images reduces to simple symbolic manipulations and testing safety and fix-point to solving a particular class of constraint satisfiability problems, called Satisfiability Modulo Theories (SMT) problems, for which scalable and efficient SMT solvers are currently available (e.g., Z3 [1]).

Enter ASASP. In [4, 3], it is studied how the MCMT approach can be used to solve (variants of) the user-role reachability problem. On the theoretical side, it is shown that the backward reachability procedure described above decides (variants of) the user-role reachability problem. On the practical side, extensive experiments have shown that an automated tool, called ASASP [2] implementing (a refinement of) the backward reachability procedure, has a good *trade-off* between *scalability* and *expressiveness*. The success of ASASP in terms of scalability is discussed in [4]: it performs significantly better than the state-of-the-art tool RBAC-PAT [8] on a set of synthetic instances of the user-role reachability problem proposed in [14]. There are two main reasons for the efficiency of ASASP: (1) the use of the Z3 SMT solver for quickly discharging the proof obligations encoding safety and fix-point tests and (2) the use of a *divide et impera* heuristics to decompose the goal of a user-role reachability problems into sub-goals (see [2] for more on this issue). The success of ASASP in terms of expressiveness is reported in [3]: it successfully solves instances of the user-role reachability problems in which role hierarchies and attributes (ranging over infinite sets of values) are used to define administrative domains that RBAC-PAT is not capable of tackling because of the following two reasons. First, the separate administration restriction (see, e.g., [14]) does not hold for the variants of the user-role reachability problem considered in [3]. Such a restriction—that distinguishes administrators from simple users—allows to solve instances of user-role reachability problems by considering only one user at a time. Second, the assumption that the cardinality of the set U of users is bounded is also not satisfied. Despite this, designers of administrative rules can still know, by using ASASP, whether security properties are satisfied or not, regardless of the number of users.

Enter MOHAWK. Immediately after ASASP, a new tool, called MOHAWK [9], has been proposed for the analysis of ARBAC policies especially tuned to error-finding rather than verification (as it is the case of both RBAC-PAT and ASASP). In [9], it is shown that MOHAWK outperforms RBAC-PAT on the problems in [14] and on a new set of much larger instances of the user-role reachability problem. It was natural to run ASASP on these new benchmark problems: rather disappointingly, it could tackle problem instances containing up to 200 roles and 1,000 administrative operations but it was unable to scale up and handle the largest instances containing 80,000 roles and 400,000 administrative operations. This is in line with the following observation of [9]: "model checking does not scale adequately for verifying policies of very large sizes." The reason of the bad scalability of ASASP can be traced back to the fact that it was designed to handle instances of the user-role reachability problems with a relatively compact but complex specification (e.g., involving attributes ranging over infinite domains). In contrast, the problem instances in [9] are quite large with very simple specifications in which there is a bounded but large number of roles, the role hierarchy is not used, and the separate administration restriction holds. (Notice that the absence of a role hierarchy is without loss of generality; see, e.g., [13].)

Exit ASASP. **Enter** MCMT. What were we supposed to do to make a tool based on the MCMT approach capable of efficiently solving the user-role reachability problem instances in [9]? One possibility was to extend ASASP with new heuristics to obtain the desired scalability. The other option was to re-use (possibly off-the-shelf) a well-engineered model checker in which to encode the user-role reachability problem for ARBAC policies. Our choice was to build a new analysis tool on top of MCMT [2], the first implementation of the MCMT approach. The advantage of this choice are twofold. First, we do not have to undergo a major re-implementation of ASASP that takes time and may insert bugs, but we only need to write a translator from instances of the user-role reachability problem to reachability problems in MCMT input language, a routine programming task. Second, we can re-use a better engineered incarnation of the MCMT approach that supports some features (e.g., the reuse of previous computations) that may be exploited to significantly improve performances, as we will see in Section 4.

MCMT **at work**. In [2], the development of ASASP is justified with the fact that it was not possible to encode user-role reachability problems in the input language of MCMT because (a) it supports only unary relations and (b) it does not allow transitions to have more than two parameters. Limitation (a) prevents the representation of the relation $UA \subseteq U \times R$ and limitation (b) does not allow to handle role hierarchies and to overcome the separate administration restriction (see [4] for a discussion about these issues). In this respect, the limited expressiveness required to specify the ARBAC policies in [9] makes the two limitations above unproblematic. Concerning (a), it is not necessary to use the binary relation UA to record user-role assignments, since the set R of roles is finite. It is sufficient to replace UA with a finite collection of sets, one per role. Formally, let $R = \{r_1, ..., r_n\}$ for $n \geq 1$, define $U_{r_i} = \{u | (u, r_i) \in UA\}$ for $i = 1, ..., n$. Straightforward modifications to the definition of $\rightarrow_\psi$ (for ψ a pair of relations *can_assign* and *can_revoke*)—given in Section 2—allows one to replace UA with the U_{r_i}'s. Concerning (b), since the role-hierarchy has been eliminated and the separate administration restriction is enforced, the definition of $\rightarrow_\psi$, for a given tuple in *can_assign* or *can_revoke*, is parametric with respect to just two users, namely the administrator and the user to which the administrative action is going to be applied. These observations enable us to use MCMT for the automated analysis of the instances of the user-role reachability problem in [9]. To this end, we have written a translator of instances of the user-role reachability problems to reachability problems expressed in MCMT input language. To keep technicalities to a minimum, we illustrate the translation on a problem from [14].

Example 1. According to [14], we consider just one user and omit administrative users and roles so that the tuples in *can_assign* are pairs composed of a simple pre-condition and a target role and the pairs in *can_revoke* reduce to target roles only. Let $U = \{u_1\}$, $R = \{r_1, ..., r_8\}$, initially $UA := \{(u_1, r_1), (u_1, r_4), (u_1, r_7)\}$, the tuples $(\{r_1\}, r_2)$, $(\{r_2\}, r_3)$, $(\{r_3, \overline{r_4}\}, r_5)$, $(\{r_5\}, r_6)$, $(\{\overline{r_2}\}, r_7)$, and $(\{r_7\}, r_8)$ are in *can_assign* whereas the elements (r_1), (r_2), (r_3), (r_5), (r_6), and (r_7) are in *can_revoke*. The goal of the problem is $(u_1, \{r_6\})$.

To formalize this problem instance in MCMT, we introduce a unary relation u_r per role $r \in R$. The initial relation UA can thus be expressed as

$$\forall x. \begin{bmatrix} u_{r_1}(x) \leftrightarrow x = u_1 \wedge u_{r_4}(x) \leftrightarrow x = u_1 \wedge u_{r_7}(x) \leftrightarrow x = u_1 \wedge u_{r_a}(x) \leftrightarrow x = u_2 \wedge \\ \neg u_{r_2}(x) \wedge \neg u_{r_3}(x) \wedge \neg u_{r_5}(x) \wedge \neg u_{r_6}(x) \end{bmatrix}.$$

For instance, $(\{r_5\}, r_6)$ in *can_assign* is formalized as

$$\exists x. \left[u_{r_5}(x) \wedge \forall y. (u'_{r_6}(y) \leftrightarrow (y = x \vee u_{r_6}(y))) \right]$$

and (r_1) in *can_revoke* as $\exists x. \left[u_{r_1}(x) \wedge \forall y. (u'_{r_1}(y) \leftrightarrow (y \neq x \wedge u_{r_1}(y))) \right]$, where u_r and u'_r indicate the value of U_r immediately before and after, respectively, of the execution of the administrative action (we also have omitted—for the sake of compactness—identical updates, i.e. a conjunct $\forall y. (u'_r(y) \leftrightarrow u_r(y))$ for each role r distinct from the target goal in the tuple of *can_assign* or *can_revoke*). The other administrative actions are translated in a similar way. The goal can be represented as $\exists x. u_{r_6}(x) \wedge x = u_1$. The pre-image of the goal with respect to $(\{r_5\}, r_6)$ is the set of states from which it is possible to reach the goal by using the administrative action $(\{r_5\}, r_6)$. This is formalized as the formula

$$\exists u'_{r_1}, ..., u'_{r_8}. (\exists x. (u'_{r_6}(x) \wedge x = u_1) \wedge \exists x. \left[u_{r_5}(x) \wedge \forall y. (u'_{r_6}(y) \leftrightarrow (y = x \vee u_{r_6}(y))) \right]),$$

that can be shown equivalent to $\exists x. u_{r_5}(x) \wedge x = u_1$ (see [4] for details). On this problem, MCMT returns `unreachable` and we conclude that $(u_1, \{r_6\})$ is unreachable, confirming the result of [14]. □

4 MCMT's new clothes for analysing ARBAC policies

The design of the techniques used to enable MCMT to scale up to handle the largest instances of the user-role reachability problem in [9] have been guided by the following two simple observations:

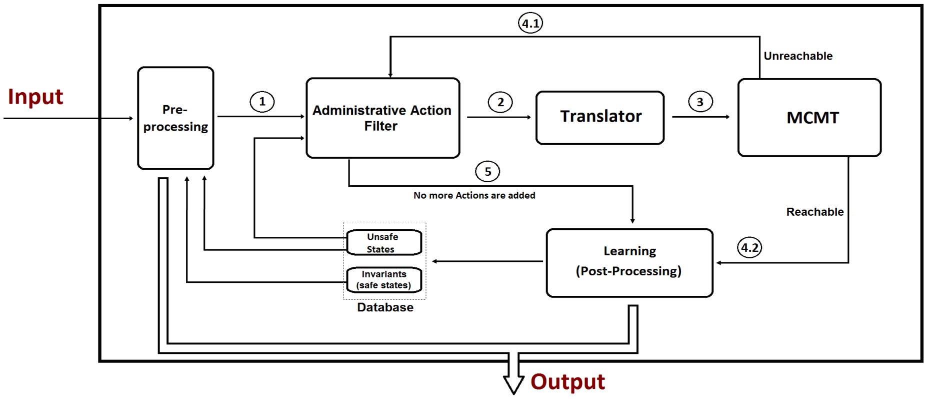

Fig. 1. ASASPXL architecture

- **(O1)** The main source of complexity is the huge number of administrative operations; thus, for scalability, the original problem must be split into smaller sub-problems by using a heuristics that tries to maximize the probability of MCMT to return `reachable`.
- **(O2)** The invocations of MCMT are computationally very expensive; thus, heuristics to minimize their numbers and reuse the findings of state space explorations of previous sub-problems to speed up the solution of newer ones are of paramount importance for scalability.

The main idea is to generate a sequence $P_0, ..., P_{n-1}, P_n$ of problem instances with a fixed goal and an increasingly larger sub-set of the administrative operations. Key to speed up the solution of problem P_{k+1} (for $0 \leq k < n$) is the capability of MCMT to reuse the information gathered when exploring the search spaces of problems $P_0, ..., P_k$. Figure 1 shows the architecture of the tool, called ASASPXL, in which we have implemented these ideas. It takes as input an instance of the user-role reachability problem (in the format of MOHAWK) and returns `reachable`, when there exists a finite sequence of administrative operations that lead from the initial RBAC policy to one satisfying the goal, and `unreachable` otherwise. We now describe the internal workings of the various modules of ASASPXL except for the **Translator** and MCMT that have already been discussed in Section 3.

4.1 Useful administrative operations

After observation (**O1**), the idea is to extract increasingly larger sub-sets of the tuples in ψ so as to generate a sequence of increasingly more precise approximations of the original instance of the user-role reachability problem. The heuristics to do this is based on the following notion of an administrative action being useful.

Definition 1. Let ψ be administrative actions and R_g a set of roles. A tuple in ψ is 0-*useful* iff its target role is in R_g. A tuple in ψ is k-*useful* (for $k > 0$) iff it is $(k-1)$-useful or its target role occurs (possibly negated) in the simple pre-condition of a $(k-1)$-useful transition. A tuple t in ψ is *useful* iff there exists $k \geq 0$ such that t is k-useful.

The set of all k-useful tuples in $\psi = (\mathit{can\_assign}, \mathit{can\_revoke})$ is denoted with $\psi^{\leq k} = (\mathit{can\_assign}^{\leq k}, \mathit{can\_revoke}^{\leq k})$. It is easy to see that $\mathit{can\_assign}^{\leq k} \subseteq \mathit{can\_assign}^{\leq k+1}$ and $\mathit{can\_revoke}^{\leq k} \subseteq \mathit{can\_revoke}^{\leq k+1}$ (abbreviated by $\psi^{\leq k} \subseteq \psi^{\leq k+1}$) for $k \geq 0$. Since the sets $\mathit{can\_assign}$ and $\mathit{can\_revoke}$ in ψ are bounded, there must exists a value $\tilde{k} \geq 0$ such that $\psi^{\leq \tilde{k}} = \psi^{\leq \tilde{k}+1}$ (that abbreviates $\psi^{\leq \tilde{k}} \subseteq \psi^{\leq \tilde{k}+1}$ and $\psi^{\leq \tilde{k}+1} \subseteq \psi^{\leq \tilde{k}}$) or, equivalently, $\psi^{\leq \tilde{k}}$ is the (least) fix-point, also denoted with $\mathit{lfp}(\psi)$, of useful tuples in ψ. Indeed, a tuple in ψ is useful iff it is in $\mathit{lfp}(\psi)$.

Example 2. Let ψ be the administrative actions in Example 1 and $R_g := \{r_6\}$. The sets of k-useful tuples for $k \geq 0$ are the following:

$$\begin{array}{ll}
\psi^{\leq 0} := (\{(\{r_5\}, r_6)\}, \{r_6\}) & \psi^{\leq 1} := \psi^{\leq 0} \cup (\{(\{r_3, \overline{r_4}\}, r_5)\}, \{r_5\}) \\
\psi^{\leq 2} := \psi^{\leq 1} \cup (\{(\{r_2\}, r_3)\}, \{r_3\}) & \psi^{\leq 3} := \psi^{\leq 2} \cup (\{(\{r_1\}, r_2)\}, \{r_2\}) \\
\psi^{\leq 4} := \psi^{\leq 3} \cup (\emptyset, \{r_1\}) & \psi^{\leq k} := \psi^{\leq 4} \text{ for } k > 4,
\end{array}$$

where $(\mathit{can\_assign}_1, \mathit{can\_revoke}_1) \cup (\mathit{can\_assign}_2, \mathit{can\_revoke}_2)$ abbreviates $(\mathit{can\_assign}_1 \cup \mathit{can\_assign}_2, \mathit{can\_revoke}_1 \cup \mathit{can\_revoke}_2)$. Notice that $\psi^{\leq 4} = \mathit{lfp}(\psi)$.

Now, consider the following instance of the user-role reachability problem: $\langle UA, \psi^{\leq 4}, (u_1, \{r_6\})\rangle$ where UA the initial user-role assignment relation in Example 1. After translation, MCMT returns `unreachable` on this problem instance. We obtain the same result if we run the tool on the translation of the following problem instance: $\langle UA, \psi^{\leq 4}, (u_1, \{r_6\})\rangle$. Interestingly, if we ask MCMT to return also the sets of user-role assignment relations that have been explored during backward reachability for the two instances (this feature of MCMT will be discussed in Section 4.3 below), we immediately realize that they are identical. This is not by accident as the following proposition shows. □

Proposition 1. A goal (u_g, R_g) is unreachable from an initial user-role assignment relation UA by using the administrative operations in ψ iff (u_g, R_g) is unreachable from UA by using the administrative operations in $\mathit{lfp}(\psi)$.

The proof of this fact consists of showing that the pre-image of the fix-point set of backward reachable states with respect to any of the administrative operations in ψ but not in $\mathit{lfp}(\psi)$ (denoted with $\psi \setminus \mathit{lfp}(\psi)$) is redundant and can thus be safely discarded. We illustrate this with an example.

Example 3. Consider again the problem instance in Example 1. The set of backward reachable states that MCMT visits during backward reachability is

$$\exists x. \left[\begin{array}{l} (u_{r_6}(x) \wedge x = u_1) \vee (u_{r_5}(x) \wedge x = u_1) \vee (u_{r_3}(x) \wedge \neg u_{r_4}(x) \wedge x = u_1) \vee \\ (u_{r_2}(x) \wedge \neg u_{r_4}(x) \wedge x = u_1) \vee (u_{r_1}(x) \wedge \neg u_{r_4}(x) \wedge x = u_1) \end{array} \right], \quad (1)$$

obtained by considering the tuples in $\psi^{\leq 4}$ only, as observed in Example 2. (It is possible to tell MCMT to save to a file the symbolic representation—such as (1)—of the set of backward reachable states visited during backward reachability.) Now, the pre-image of (1) with respect to $(\{r_7\}, r_8) \in \psi \setminus \psi^{\leq 4}$ is the formula $(1) \wedge \exists x.u_{r_7}(x)$ as r_8 does not occur in (1). Indeed, such a formula trivially implies (1) (or, equivalently, the conjunction of (1) with the negation of $(1) \wedge \exists x.u_{r_7}(x)$ is unsatisfiable) and the fix-point test is successful, confirming that (1) is also a fix-point with respect to the administrative operations in $\psi^{\leq 4} \cup \{(\{r_7\}, r_8)\}$. Similar observations hold for the other tuples in $\psi \setminus \psi^{\leq 4}$ allowing us to conclude that (1) is also a fix-point with respect to ψ. □

A formal proof of the proposition can be obtained by adapting the framework in [4] to the slightly different symbolic representation for ARBAC policies adopted in this paper.

The module **Administrative action filter** in Figure 1 uses the notion of k-useful tuple to build a sequence of increasingly precise instances of user-role reachability problem. Such a sequence is terminated either when the goal is found to be reachable or when the fix-point of useful administrative operations is detected (by Proposition 1, this is enough to conclude that a goal is unreachable with respect to the whole set of administrative operations). Given an instance $\langle UA, \psi, (u_g, R_g)\rangle$ of the user-role reachability problem, the **Administrative action filter** works as follows:

```
1. Let $k := 0$ and $UT$ be the set of $k$-useful tuples in $\psi$
2. Repeat
   (a) Translate the instance $\langle UA, UT, (u_g, R_g)\rangle$ of the user-role reachability
       problem to MCMT input language
   (b) If MCMT returns reachable, then return reachable
   (c) Let $k := k + 1$, $PUT := UT$, and $UT$ be the set of $k$-useful tuples in $\psi$
3. Until $PUT = UT$
4. Return unreachable
```

Initially, UT contains $\psi^{\leq 0}$. At iteration $k \geq 1$, UT stores $\psi^{\leq k}$ and PUT contains $\psi^{\leq (k-1)}$. For $k \geq 0$, the instance $\langle UA, \psi^{\leq k}, (u_g, R_g)\rangle$ of the user-role reachability problem is translated to MCMT input language (step 2(a)). In case MCMT discovers that the goal (u_g, R_g) is reachable with the sub-set $\psi^{\leq k}$ of the administrative operations, *a fortiori* (u_g, R_g) is reachable with respect to the whole set ψ, and the module returns (step 2(b)). Otherwise, a new instance of the user-role reachability problem is considered at the next iteration if the condition at step 3 does not hold, i.e. PUT does not yet store $\mathit{lfp}(\psi)$. If the condition at step 3 holds, by Proposition 1, we can exit the loop and return the unreachability of the goal with respect to the whole set ψ of administrative operations. The termination of the loop is guaranteed by the existence of $\mathit{lfp}(\psi)$. Notice how two instances of the user-role reachability problem at iterations k and $k + 1$ only differ for the administrative actions in $\psi^{\leq (k+1)} \setminus \psi^{\leq k}$ while they share those in $\psi^{\leq k}$ since $\psi^{\leq k} \subseteq \psi^{\leq (k+1)}$.

4.2 Reducing the number of invocations to the model checker

Recall the first part of observation (**O2**) that suggests to find ways to reduce the number of invocations to MCMT. Our idea is to exploit two interesting capabilities of MCMT: (a) saving (to a file) the symbolic representation of the state space explored when the goal is unreachable and (b) returning a sequence of transitions that lead from the initial state to a state satisfying the goal.

The crucial observation to exploit capability (a) is that the negation of the formula representing the (fix-point) set F of backward reachable states (e.g., formula (1) in Example 3 above) is the (strongest) invariant whose intersection with the set G of states satisfying the goal is empty (see [7] for more on this point). The negation of the formula representing F (together with the other components of the instance of the user-role reachability problem that has generated it) is stored by the **Learning** (**Post-Processing**) module to the database labelled **Invariants** in Figure 1 and it is used in the module **Pre-processing** (see Figure 1) as follows. Assume that a new instance of the user-role reachability problem shares the same initial user-role assignment relation and the same set of administrative operations associated to a formula φ stored in the database **Invariants**. If the formula representing the new goal is such that in conjunction with φ is unsatisfiable, then we can immediately conclude that also the new goal is unreachable. We illustrate this with an example.

Example 4. Consider again the instance of the user-role reachability problem in Example 1 and formula (1) representing the the symbolic representation of the set of backward reachable states that have been visited during backward reachability. The conjunction of (1) with that representing the initial relation UA (reported in Example 1) is unsatisfiable (safety check) and MCMT returns `unreachable` (as anticipated in Example 1). At this point, the negation of (1), i.e.

$$\forall x. \begin{bmatrix} (x = u_1 \to \neg u_{r_6}(x)) \wedge (x = u_1 \to \neg u_{r_5}(x)) \wedge (x = u_1 \wedge u_{r_3}(x) \to u_{r_4}(x)) \wedge \\ (x = u_1 \wedge u_{r_2}(x) \to u_{r_4}(x)) \wedge (x = u_1 \wedge u_{r_1}(x) \to u_{r_4}(x)) \end{bmatrix} \quad (2)$$

is stored in the database **Invariants** together with the initial user-role assignment relation and administrative operations of Example 1.

Now, consider that the next instance of the user-role reachability problem to solve is composed of the same initial user-role assignment relation and administrative operations and the goal is $(u_1, \{r_5\})$. Since the conjunction of the symbolic representation of the goal, $\exists x.(x = u_1 \wedge u_{r_5}(x))$, with (2) is obviously unsatisfiable (this can be quickly established by an available SMT solver), our system immediately returns `unreachable` also for this instance without invoking MCMT. □

Now, we turn to the problem of exploiting capability (b) of MCMT, i.e. returning a sequence σ of transitions leading from the initial state to a state satisfying the goal. For this, notice that each state generated by applying a transition in σ is indeed also reachable. The symbolic representation G of the sets of user-role assignment relations generated by the application of the administrative operations in σ (together with the initial user-role assignment relation and the sequence

σ) are stored by the **Learning** (**Post-Processing**) module to the database labelled **Unsafe States** in Figure 1 and are used by the module **Pre-processing** (see again Figure 1) as follows. Assume that a new instance of the user-role reachability problem shares the same initial user-role assignment relation and contains at least the administrative operations in σ associated to the sequence γ of user-role assignment relations generated by the applications of the operations in σ. If the goal g of a new problem instance is in γ, then we can immediately conclude that g is reachable. We illustrate this with an example.

Example 5. Consider the following instance of the user-role reachability problem: $\langle UA, \psi, (u_1, \{r_2, r_8\})\rangle$, where UA and ψ are those of Example 1. On the translated reachability problem, MCMT returns `reachable` with the following sequence $\sigma = (\{r_7\}, r_8); (\{r_1\}, r_2)$ of administrative operations. The sequence γ of states obtained by computing the pre-image of the goal with respect to the administrative operations in σ contains the goal itself $g_0 := \exists x.(x = u_1 \wedge u_{r_2}(x) \wedge u_{r_8}(x))$, the pre-image of g_0 with respect to $(\{r_1\}, r_2)$, i.e. $g_1 := \exists x.(x = u_1 \wedge u_{r_1}(x) \wedge u_{r_8}(x))$, and the pre-image of g_1 with respect to $(\{r_7\}, r_8)$, i.e. $\exists x.(x = u_1 \wedge u_{r_1}(x) \wedge u_{r_7}(x))$. This information is stored in the database **Unsafe States**.

If we now consider the following instance of the user-role reachability problem: $\langle UA, \psi, (u_1, \{r_1, r_8\})\rangle$, where UA and ψ are again as in Example 1, ASASPXL immediately returns `reachable` without invoking MCMT because the symbolic representation of this goal is equal to g_1 in the database **Unsafe States**. □

This concludes the description of the internal workings of the **Pre-processing** module in Figure 1 that tries to minimize the number of invocations of MCMT (first part of observation (**O2**)). The description of the **Learning** (**Post-processing**) module will be finished in the following (sub-)section.

4.3 Reusing previously visited states

Recall the second part of observation (**O2**) that suggests to re-use as much as possible the results of previous invocations of the model checker. Our idea is to save the sets of user-role assignment relations visited when solving the instance $P_k = \langle UA, \psi^{\leq k}, (u_g, R_g)\rangle$ of the user-role reachability problem generated by the **Administrative action filter** module (see Section 4.1) so as to avoid to visit them again when solving the next instance $P_{k+1} = \langle UA, \psi^{\leq (k+1)}, (u_g, R_g)\rangle$. As observed above (see the last sentence of Section 4.1), two successive instances P_k and P_{k+1} of the user-role reachability problem generated by the **Administrative action filter** module only differ for the administrative actions in $\psi^{\leq (k+1)} \setminus \psi^{\leq k}$ and share those in $\psi^{\leq k}$. When solving P_{k+1}, it would thus be desirable to visit only the states generated by the actions in $\psi^{\leq (k+1)} \setminus \psi^{\leq k}$ and avoid to recompute those generated by the actions in $\psi^{\leq k}$, that have already been visited when solving P_k.

The description of the missing part of the internal workings of the **Learning** (**Post-processing**) module can be completed as follows. Consider sub-problem $P_k = \langle UA, \psi^{\leq k}, (u_g, R_g)\rangle$ for $k \geq 0$. There are two cases to consider depending on the fact that (u_g, R_g) is reachable or not.

First, assume that MCMT has found (u_g, R_g) to be unreachable and that φ is the formula representing the complement of the set of backward reachable states. Before solving sub-problem P_{k+1}, the **Learning** module deletes from $\psi^{\leq(k+1)}$ the administrative actions whose symbolic representation of the (simple) pre-condition implies φ. The correctness of doing this is stated in the following proposition.

Proposition 2. Let $\langle UA, \psi, (u_g, R_g)\rangle$ be an instance of the user-role reachability problem such that (u_g, R_g) is unreachable and F is the set of backward reachable states. If $t \not\in \psi$ is an administrative operation whose simple pre-condition is contained in F, then (u_g, R_g) is also unreachable when considering the instance $\langle UA, \psi \cup \{t\}, (u_g, R_g)\rangle$ of the user-role reachability problem.

The proof of this fact is based on the following two observations. First, if t is not useful then we can safely ignore it by Proposition 1. Second, if t is useful then the pre-image of F with respect to t is contained in F because, by assumption, its simple pre-condition is contained in F. We illustrate this with an example.

Example 6. Consider the instance of the user-role reachability problem in Example 1. The goal is unreachable and the set of backward reachable states F is symbolically represented as (1). Consider $t_1 = (\{r_2, \overline{r_4}\}, r_8)$: this is not useful, the pre-image of F with respect to t_1 is redundant by Proposition 1, F is a fix-point also with respect to $\psi \cup \{t_1\}$, and the goal is still unreachable. Consider now $t_2 = (\{r_2, \overline{r_4}\}, r_3)$: the pre-image of F with respect to t_2 is symbolically represented by the formula $\exists x.(u_{r_2}(x) \wedge \neg u_{r_4}(x) \wedge x = u_1)$. An available SMT solver easily shows that this formula implies (1), F is a fix-point also with respect to $\psi \cup \{t_2\}$, and the goal is still unreachable. □

The second case of operation for the **Learning** module is when MCMT finds (u_g, R_g) to be reachable by a sequence $\sigma = t_1; \cdots; t_n$ of administrative operations. Let $g_1; \cdots; g_n$ be the sequence of user-role assignment relations obtained by applying t_j to g_{j-1} for $j = 1, ..., n$ with $g_0 = UA$. Before solving a new instance of the user-role reachability problem with the same initial user-role assignment relation UA and whose administrative actions contain those in σ, the **Learning** module adds to $\psi^{\leq(k+1)}$ the transition having as (simple) "pre-condition" UA and as "update" g_j, for $j = 2, ...n$. (Notice that the additional transitions do not enlarge the set of reachable states.) We illustrate this with an example.

Example 7. Consider again the first instance of the user-role reachability problem of Example 5: $\langle UA, \psi, (u_1, \{r_2, r_8\})\rangle$ where UA and ψ are those in Example 1. As already said, the goal is reachable with the sequence $\sigma = (\{r_7\}, r_8); (\{r_1\}, r_2)$. Thus, the **Learning** module would add the following (redundant) transition:

$$\exists x. \begin{pmatrix} u_{r_1}(x) \wedge u_{r_4}(x) \wedge u_{r_7}(x) \wedge \\ \forall y.(u'_{r_2}(y) \leftrightarrow (y = x \vee u_{r_2}(y))) \wedge \forall y.(u'_{r_8}(y) \leftrightarrow (y = x \vee u_{r_8}(y))) \end{pmatrix},$$

where identical updates have been omitted for the sake of simplicity.

Consider now the following new instance of the user-role reachability problem: $\langle UA, \psi, (u_1, \{r_3\})\rangle$. It is immediate to see that the goal is reachable by the sequence $\sigma' = \sigma; (\{r_2\}, r_3)$ of administrative operations. Because of the availability of the additional transition above—whose execution has the same effect of the (atomic) sequential execution of the administrative actions in σ—the reachability of the goal can be detected in two steps instead of three. □

As illustrated in the example, the hope is that establishing the reachability (if the case) of the goal of a new instance of the user-role reachability problem could be done by using one of the additional transitions whose effects is equivalent to the execution of several transitions, thereby speeding up the search procedure.

This concludes the description of the internal workings of the **Learning** (**Post-processing**) module in Figure 1.

4.4 Putting things together

We now describe the flow of execution among the various modules in ASASPXL (see Figure 1). The input instance $P = \langle UA, \psi, (u_g, R_g)\rangle$ of the user-role reachability problem is given to the **Pre-processing** module that searches the databases **Unsafe states** and **Invariants** (**Safe states**) to see whether the goal can be declared `reachable` or `unreachable` because of the cached results of previous invocations to the model checker (Section 4.2). If no previous information allows us to conclude, the instance is passed (arrow labelled with 1) to the **Administrative Action Filter** that computes a sequence $P_1, ..., P_k$ of increasingly precise approximations of P by using the notion of useful administrative operations (Section 4.1). Each P_k is sent (arrow labelled with 2) to the **Translator** that converts the problem instance to a reachability problem in MCMT input language. At this point, the model checker is invoked (arrow labelled with 3) on the resulting problem and two outcomes are possible. If the goal is unreachable, then control is given back to the **Administrative Action Filter** (arrow labelled 4.1) that considers a more precise approximation of the problem (if any) that is translated and solved again by MCMT. If this is not possible, control is passed to the **Learning** module (arrow labelled 5) that declares the instance of the user-role reachability problem to be `unreachable` and updates the database **Invariants** (Section 4.2). Instead, if the goal is found reachable by MCMT, then control is passed directly to the **Learning** module (arrow labelled 4.2) that declares the instance of the user-role reachability problem to be `reachable` and updates the database **Unsafe states** (Section 4.2).

The completeness of ASASPXL derives from the properties discussed in Sections 4.1, 4.2, and 4.3 as well as the correctness of the encoding in the module **Translation** (see end of Section 3).

5 Experiments

We have implemented ASASPXL in Python and have conducted an exhaustive experimental evaluation to compare it with MOHAWK on the set of "Complex Policies" in [9], composed of three synthetic test suites: **Test suite 1**—whose problem instances can be solved in polynomial time, **Test suite 2**—that are NP-complete, and **Test suite 3**—that are PSPACE-complete. We do not consider the “Simple Policies” in [9] as their solving time is very low and are not suited to evaluate the scalability of the tools. We do not consider other tools (e.g., RBAC-PAT [8]) as the experiments in [9] clearly shows that MOHAWK is superior. For example, RBAC-PAT when performing backward reachability is reported to seg-fault on problem instances containing at least 20 roles and 100 rules in all test suites while it is significantly slower than MOHAWK when run in forward

Table 1. Experimental results on the “complex” benchmarks in [9]

Test suite	**# Roles, # Rules**	MOHAWK (Slicing time + Verification time)	ASASPXL	**Variance**	
				MOHAWK	ASASPXL
Test suite 1	3, 15	**0.42** (0.17 + 0.25)	0.12	0.00034	0.00126
	5, 25	**0.50** (0.20 + 0.30)	0.22	0.00104	0.02188
	20, 100	**0.60** (0.28 + 0.32)	0.11	0.00048	0.00314
	40, 200	**0.94** (0.39 + 0.55)	0.10	0.19242	0.00294
	200, 1000	**2.65** (1.25 + 1.40)	0.18	0.7027	0.02758
	500, 2500	**4.87** (2.27 + 2.60)	0.43	7.0337	0.29594
	4000, 20000	**16.90** (11.41 + 5.49)	1.64	1.26694	0.11166
	20000, 80000	**71.56** (44.70 + 26.86)	24.17	7.56264	0.27724
	30000, 120000	**195.54** (119.39 + 76.15)	59.08	66.4833	0.38058
	40000, 200000	**455.14** (263.82 + 191.32)	109.07	32.35406	2.42496
	80000, 400000	**2786.33** (1600.22 + 1186.11)	398.63	1251.832	0.51542
Test suite 2	3, 15	**0.40** (0.16 + 0.24)	0.12	0.00046	0.00204
	5, 25	**0.50** (0.19 + 0.31)	0.21	0.0019	0.02012
	20, 100	**0.54** (0.25 + 0.29)	0.10	0.00036	0.00242
	40, 200	**1.21** (0.37 + 0.84)	0.10	1.07136	0.00108
	200, 1000	**2.54** (1.24 + 1.30)	0.14	0.6452	0.01008
	500, 2500	**5.02** (2.29 + 2.73)	0.43	5.91882	0.32836
	4000, 20000	**14.33** (9.65 + 4.68)	1.48	0.53058	0.06206
	20000, 80000	**74.32** (45.35 + 28.97)	24.99	13.9347	0.0716
	30000, 120000	**194.85** (115.58 + 79.27)	57.09	42.39056	0.18292
	40000, 200000	**470.89** (262.39 + 208.50)	98.49	585.6608	0.26196
	80000, 400000	**2753.12** (1589.97 + 1163.15)	360.96	1493.596	3.19596
Test suite 3	3, 15	**0.41** (0.17 + 0.24)	0.09	0.00012	0.00078
	5, 25	**0.47** (0.19 + 0.28)	0.08	0.00164	0.0001
	20, 100	**0.77** (0.29 + 0.48)	0.54	0.0771	0.08822
	40, 200	**0.77** (0.38 + 0.39)	0.37	0.0012	0.00468
	200, 1000	**5.93** (1.53 + 4.4)	1.51	47.2814	0.20348
	500, 2500	**3.78** (2.05 + 1.73)	1.12	0.05662	0.00298
	4000, 20000	**14.05** (9.96 + 4.09)	11.13	0.09255	0.317425
	20000, 80000	**80.61** (48.64 + 31.97)	27.25	23.98093	2.974775
	30000, 120000	**259.15** (148.35 + 110.80)	97.55	325.5216	6343.912
	40000, 200000	**604.17** (346.10 + 258.07)	110.65	1247.141	110.9948
	80000, 400000	**3477.19** (1951.41 + 1525.78)	402.22	2776.703	0.50856

reachability mode; e.g., according to [9], the instances with 20,000 roles and 80,000 rules in the three test suites are solved in around a minute by Mohawk while rbac-pat goes in time out after 60 minutes.

For each set of ARBAC policies in the test suites, we have generated an instance of the user-role reachability problem by considering an empty initial user-role assignment relation and 5 distinct (randomly generated) goals that are selected to be reachable (this kind of instances of the user-role reachability problem are said to be instances of the error-finding problem in [9]).

Table 1 reports the results of running asaspXL and Mohawk on these instances. Column 1 reports the name of the test suite, column 2 contains the number of roles and administrative operations in the policy, column 3 and 4 the average times (in seconds) taken by Mohawk and asaspXL, respectively, to solve the five instances of the user-role reachability problem associated to an ARBAC policy, and column 4 the variance in solving times for Mohawk and asaspXL, respectively. For Mohawk, in column 3 we report also the time spent in the slicing phase (a technique for eliminating irrelevant users, roles, and administrative operations that are non relevant to solve a certain instance of the user-role reachability problem, see [9, 14] for more details) and the verification phase (i.e. the abstract-check-refine model checking technique described in [9]). All experiments were performed on an Intel Core 2 Duo T6600 (2.2 GHz) CPU with 2 GB Ram running Ubuntu 11.10.[3]

The results clearly show that asaspXL performs significantly better than Mohawk. In many cases, asaspXL overall time is less than Mohawk Verification time (see column 3), i.e. even disregarding the Slicing time. Furthermore, the behaviour of asaspXL is more predictable than Mohawk since the variance of the latter is much larger (see the last column of the table). The results also demonstrate the effectiveness of our approach and nicely complement the theoretical properties discussed in Section 4 that aim to guarantee that asaspXL will not miss errors (if any). This is in contrast with Mohawk, that, as said in [9], is incomplete and thus may not find all errors.

6 Conclusions

We have presented techniques to enable a model checker to solve large instances of the user-role reachability problem for ARBAC policies. The model checker is assumed to provide some basic functionalities such as the capability of storing the already visited sets of states for later reuse and that of returning the sequence of transitions that lead from an initial state to a state satisfying the goal (if the case). In our implementation, we have used mcmt but any model checker with these features can be plugged in. (This is so because we work under the separate administration restriction and the capability of mcmt to handle infinite state systems is not used since—as observed in, e.g., [14]—it is possible to consider just

[3] We thank the authors of Mohawk for making available to us the latest version of their tool. We also thank the support team of NuSMV for their help with the installation of the tool, a necessary pre-requisite for using Mohawk.

one administrator without loss of generality.) We have shown that the proposed techniques do not miss errors in buggy policies; this is in contrast with MOHAWK that is incomplete. We have also provided evidence that an implementation of the proposed techniques, called ASASPXL, performs significantly better than MOHAWK on the larger problem instances in [9].

As future work, we plan to design and implement ASASP 2.0, a tool that combines the flexibility of ASASP [2] with the scalability of ASASPXL. To this end, we are currently collecting a database of heterogeneous problem instances that will help us to understand the right level of expressiveness. In this respect, the benchmark problems recently proposed in [6] (together with those in [3]) will be of particular interest since the analysis must be done with respect to a finite but unknown number of users. For such problems, the capability of MCMT to handle infinite state systems will be key.

Acknowledgements. This work was partially supported by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7).

References

1. http://research.microsoft.com/en-us/um/redmond/projects/z3.
2. F. Alberti, A. Armando, and S. Ranise. ASASP: Automated Symbolic Analysis of Security Policies. In *CADE*, volume 6803 of *LNCS*, pages 26–34. Springer, 2011.
3. F. Alberti, A. Armando, and S. Ranise. Efficient Symbolic Automated Analysis of Administrative Role Based Access Control Policies. In *ASIACCS*. ACM Pr., 2011.
4. A. Armando and S. Ranise. Automated Symbolic Analysis of ARBAC Policies. In *6th STM Workshop*, volume 6710 of *LNCS*, pages 17–33. Springer, 2010.
5. J. Crampton. Understanding and developing role-based administrative models. In *Proc. 12th CCS, pages 158–167, ACM Press, 2005*.
6. A. L. Ferrara, P. Madhusudan, and G. Parlato. Security Analysis of Access Control Policies through Program Verification. In *CSF*, 2012.
7. S. Ghilardi and S. Ranise. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. *In LMCS, Vol. 6, Issue 4*, 2010.
8. M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller. Rbacpat: A policy analysis tool for role based access control. In *TACAS*, volume 5505 of *LNCS*, pages 46–49. Springer, 2009.
9. K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin. Automatic Error Finding for Access-Control Policies. In *CCS*. ACM, 2011.
10. S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and H. Winsborough. Towards formal verification of role-based access control policies. *IEEE TDSC*, 5(4):242–255, 2008.
11. N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM TISSEC*, 9(4):391–420, 2006.
12. R. Sandhu, E. Coyne, H. Feinstein, and C. Youmann. Role-Based Access Control Models. *IEEE Computer*, 2(29):38–47, 1996.
13. A. Sasturkar, P. Yang, S. D. Stoller, and C.R. Ramakrishnan. Policy analysis for administrative role based access control. In *CSF*. IEEE Press, July 2006.
14. S. D. Stoller, P. Yang, C.R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS*. ACM Press, 2007.